lecture 4

Brad McNeney

2017-01-25

Data management

Data management overview

- After reading data into a data frame, we need to manage it:
  - print, view and edit data frames, add/delete variables
  - derive new variables from old
  - merge and reshape datasets
- There are many tools in “base” R.
- Recently the `dplyr` package has become popular for management of data frames.
  - Design goal is to make data management more intuitive
  - Will discuss `dplyr` alternatives where possible.

Print, view, edit

print(), View() and edit()

- print() prints R objects
  - This function is "generic", meaning that it will try to find the specific function to print specific objects (e.g., print.data.frame).
- View() launches a new window (or RStudio tab) to view a data frame and edit() launches a data editor.

Access variables in a data frame

- Can use what we learned about subsetting:

```
testdf = data.frame(ID=1:3,age=c(8,11,14),height=c(52,63,70))
testdf$ratio <- testdf$height/testdf$age
```

- or can use `with()`

```
testdf$ratio <- with(testdf,height/age)
```

- Notice how we can add a new variable to `testdf` by assignment.

Using attach() to attach a data frame

- What with() is doing is (i) create a temporary environment, (ii) copy the variables in testdf into this environment, (iii) evaluate the expression height/age in this temporary environment and (iv) return the results.
- We can do this manually with attach(testdf) followed by the expression/assignment, and then use detach("testdf") to get rid of the temporary environment.
- However, the original data frame and its copy can get out of sync and cause confusion, or we might forget to detach().
- Attaching data frames is generally frowned upon.

Adding and deleting variables from a data frame

- We saw how $ can be used to add variables to a data frame.
- Remove variables by setting to NULL

```
testdf$ratio <- NULL
testdf
```

```
##   ID age height
## 1  1   8     52
## 2  2  11     63
## 3  3  14     70
```

- To rename a variable, can add it under new name and remove old variable. Also see the `rename()` function in the `dplyr` package.

Derived variables

Adding derived variables with `tranform()`

- `transform()` together with an assignment can add and/or modify variables:

```r
testdf <- transform(testdf,ID = c("E","K","H"),
                    ratio = height/age)
testdf
```

```
##   ID age height    ratio
## 1  E   8     52 6.500000
## 2  K  11     63 5.727273
## 3  H  14     70 5.000000
```

Adding derived variables with within()

- within() is similar to transform() but allows us to use variables we create in the call:

```
testdf <- within(testdf, {
  heightcm <- height*2.54 # now use new variable heightcm
  ratiocm <- heightcm/age
  })
testdf
```

```
##   ID age height    ratio  ratiocm heightcm
## 1  E   8     52 6.500000 16.51000   132.08
## 2  K  11     63 5.727273 14.54727   160.02
## 3  H  14     70 5.000000 12.70000   177.80
```

Adding derived variables with mutate()

- mutate() from the dplyr package is very similar to transform()
  - Being from dplyr suggests using the forward pipe %>% to chain multiple mutate()s

```
library(dplyr)
testdf %>%
  select(ID,age,height) %>%
  mutate(heightcm = height*2.54) %>%
  mutate(ratiocm = heightcm/age) -> testdf
testdf
```

```
##   ID age height heightcm  ratiocm
## 1  E   8     52   132.08 16.51000
## 2  K  11     63   160.02 14.54727
## 3  H  14     70   177.80 12.70000
```

- Notice the use of -> to assign the results of our data manipulations.

Creating and working with categorical variables

- We may want to
  - create categorical by binning a numeric
  - create categorical with logical conditions
  - recode categories

Binning a numeric variable with cut()

- Creates a factor based on equal-width bins by default:

```r
set.seed(1)
n <- 100
age <- sample(17:85,size=n,replace=TRUE)
agecat <- cut(age,breaks=5)
table(agecat)
```

```
## agecat
## (16.9,30.6] (30.6,44.2] (44.2,57.8] (57.8,71.4] (71.4,85.1]
##          15          23          18          26          18
```

DIY bins with cut()

- Custom bins. Be careful not not to exclude any data values.

```
agecat <- cut(age,breaks = c(17,30,40,50,60,70,80))
agecat[age==17]
```

```
## [1] <NA>
## Levels: (17,30] (30,40] (40,50] (50,60] (60,70] (70,80]
```

```
agecat <- cut(age,breaks = c(15,30,40,50,60,70,80))
table(agecat)
```

```
## agecat
## (15,30] (30,40] (40,50] (50,60] (60,70] (70,80]
##      15      16      19       9      20      17
```

Create categorical from logical conditions

- Usual strategy is to initialize a vector to a baseline category and then use logical conditions to assign category of subsets.

```r
group <- sample(c("A","B"),size=n,replace=TRUE)
catvar <- rep(1,n)
catvar[age<50 & group=="A"] <- 2
catvar[age<60 & group=="B"] <- 3
table(catvar)
```

```
## catvar
##  1  2  3
## 49 20 31
```

Recoding variables

- For numeric or character categories use logical conditions.

```
catvar[catvar==3] <- 11 # 11 gets recycled
```

- For factors, remember that they are numeric with character labels, or levels – just change the levels

```
head(agecat)
```

```
## [1] (30,40] (40,50] (50,60] (70,80] (15,30] (70,80]
## Levels: (15,30] (30,40] (40,50] (50,60] (60,70] (70,80]
```

```
levels(agecat)[1] <- "[17,30]"
head(agecat)
```

```
## [1] (30,40] (40,50] (50,60] (70,80] [17,30] (70,80]
## Levels: [17,30] (30,40] (40,50] (50,60] (60,70] (70,80]
```

Using recode() and recode_factor() from dplyr

- Can recode multiple values at once and use with %>%

```
# Enclose numeric values in backticks
head(recode(catvar,`1`="pen",`2`="pineapple",`11`="apple"))
```

```
## [1] "apple"     "pineapple" "pen"       "pen"       "apple"     "pen
```

```
cut(age,breaks = c(15,30,40,50,60,70,80)) %>%
  recode_factor("(15,30]" = "[17,30]", "(70,80]" = "(70,100]") %>%
  head()
```

```
## [1] (30,40]  (40,50]  (50,60]  (70,100] [17,30]  (70,100]
## Levels: [17,30] (70,100] (30,40] (40,50] (50,60] (60,70]
```

- Notice how the order of the levels has changed.

Dates

- We have seen the as.Date() function for coercing character strings to Date objects.
  - The function first tries the format yyyy-mm-dd, then yyyy/mm/dd.
- Summary functions such as mean() and diff() can handle Date objects.

```
dd <- c("2002-04-02","2005-08-17","2008-08-12")
dd <- as.Date(dd)
mean(dd)
```

```
## [1] "2005-06-30"
```

```
diff(dd)
```

```
## Time differences in days
## [1] 1233 1091
```

Reading dates in other formats

- If your dates are in a format other than yyyy-mm-dd or yyyy/mm/dd you will have to specify.
- The formatting rules are described in `help(strptime)`.

```
dd <- c("05/14/1966","04/02/2002","08/17/2005","08/12/2008")
dd <- as.Date(dd,format = "%m/%d/%Y")
dd
```

```
## [1] "1966-05-14" "2002-04-02" "2005-08-17" "2008-08-12"
```

```
mean(dd)
```

```
## [1] "1995-09-18"
```

```
diff(dd)
```

```
## Time differences in days
## [1] 13107  1233  1091
```